UNIFESP
UNIVERSIDADE FEDERAL DE SÃO PAULO
1933
Instituto de Ciência e Tecnologia
Universidade Federal de São Paulo

Compiladores:
Geração de Código Intermediário

Profª Thaína A. A. Tosta

tosta.thaina@unifesp.br

São José dos Campos – 2021/2

Geração de Código Intermediário

- A geração de código intermediário é a 1ª etapa da fase de síntese do compilador;
- O código intermediário é independente da máquina alvo;
- O código intermediário é uma "linearização" da árvore sintática.

Geração de Código Intermediário

- O código intermediário é preferível, como representação intermediária, do que a árvore sintática

  Se aproxima mais da representação do código de montagem *(assembly)* com o conjunto de instruções do processador (ADD, SUB, CMP, CALL, LOAD, STORE, MOVE, AND, OR, JMP...);

- Representações intermediárias mais conhecidas:
  - Código de três endereços
  - P-código

Geração de Código Intermediário

Código de três endereços

$$x = y \; op \; z$$

- Possibilita especificar instruções com no máximo três operandos (variáveis ou constantes);
- Os tipos básicos de instruções são:
  - Expressão com atribuição;
  - Desvio;
  - Invocação (e definição) de sub-rotinas;
  - Acesso indexado.

Geração de Código Intermediário

Código de três endereços: instruções de atribuição

São aquelas em que o resultado de uma operação é armazenado na variável especificada à esquerda do operador de atribuição, denotado por =

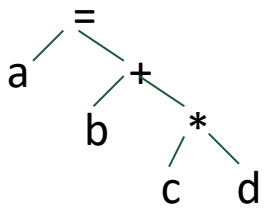
- Formas básicas de atribuição:

le = ld	ex: a = b
le = ld1 op ld2	ex: a = b + c
le = op ld	ex: a = -b

Geração de Código Intermediário

Código de três endereços: instruções de atribuição

- Expressões de atribuição mais complexas demandam a criação de variáveis temporárias, e a “quebra” em múltiplas instruções;
- Variáveis temporárias devem ter rótulos diferentes dos identificadores do programa fonte.
  - Ex: a = b + c * d




```
_t1 = c * d
_t2 = b + _t1
a = _t2
```

Geração de Código Intermediário

Código de três endereços: instruções de desvio

- Podem assumir duas formas básicas:
  - Desvio incondicional, no formato:

    goto L

    onde *L* é um rótulo que identifica uma linha de código
  - Desvio condicional, no formato:

    if x opr y goto L

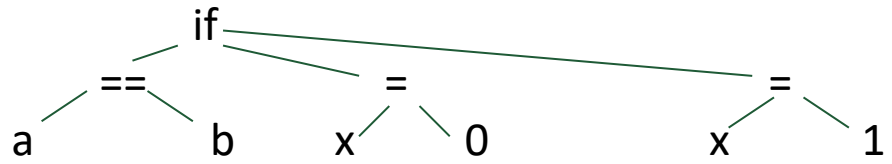
    onde *opr* é um operador relacional e *L* é o rótulo da linha que deve ser executada se o resultado da condição for verdadeiro, caso contrário, a linha seguinte é executada.

Geração de Código Intermediário

Código de três endereços: instruções de desvio

Exemplo (em C-):

```
if  (a == b) x = 0;
else x = 1;
```



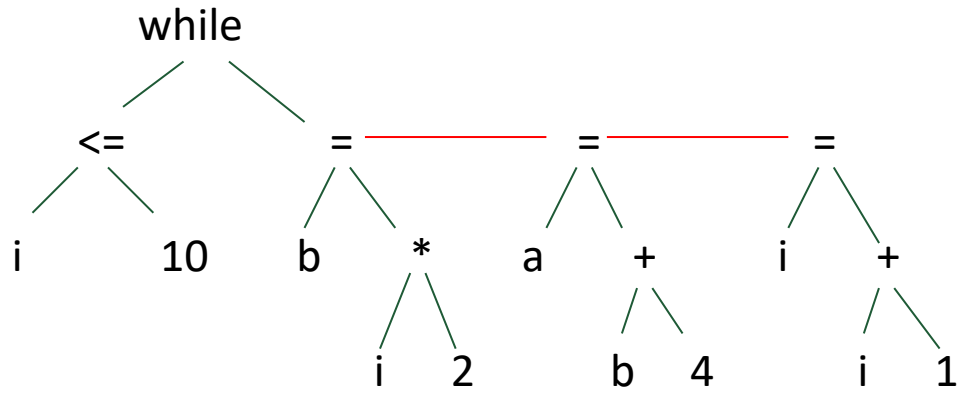

```
        If  a == b  goto  _L1   (salto para bloco "verdadeiro")
        x = 1                   (bloco "falso")
        goto  _L2               (salto para próxima instrução)
_L1:    x = 0                   (bloco "verdadeiro")
_L2:    ...                     (próxima instrução)
```

Geração de Código Intermediário

Código de três endereços: instruções de desvio

Exemplo (em C-):

```
while (i <= 10)
{
   b = i * 2;
   a = b + 4;
   i = i + 1;
}
```




```
_L1:    if  i  >  10    goto  _L2
        b = i * 2
        a = b + 4
        i = i + 1
        goto  _L1
_L2:    ...                          (próxima instrução)
```

Geração de Código Intermediário

Código de três endereços: invocação de sub-rotinas

- Assume o seguinte formato

  param x_1
  param x_2
  ⋮
  param x_n
  call *f*, *n*

  onde *f* é o nome da função e *n* é a quantidade de parâmetros de entrada da função.

- *call* é o nome da instrução para ativação da função (código *assembly* normalmente usa esse nome).

  Obs: o parâmetro de retorno da função, se houver, deve ser associado a uma atribuição.

Geração de Código Intermediário

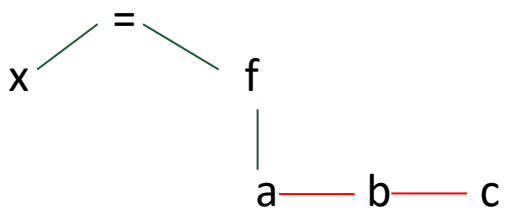
Código de três endereços: invocação de sub-rotinas

Exemplo (em C-):

```
x = f(a, b, c);
```

Código de três endereços:

```
param a
param b
param c
x = call f, 3
```



Geração de Código Intermediário

Código de três endereços: invocação de sub-rotinas

Exemplo (em C-):

a = g (b, h(c));

Código de três endereços:

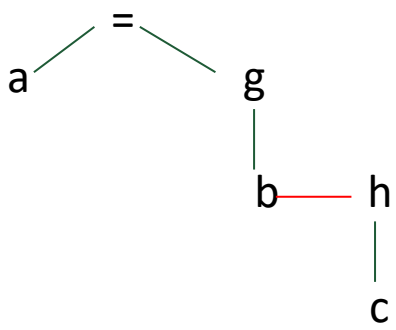
param b

param c

_t1 = call h, 1

param _t1

a = call g, 2



Obs: os parâmetros anteriores, já consumidos, devem ser desconsiderados para a próxima ativação de sub-rotina.

Geração de Código Intermediário

Código de três endereços: definição de sub-rotinas

- Para uma sub-rotina ser invocada, ela precisa ter uma definição;
- A definição assume o formato:

  L:

  (corpo da sub-rotina)

  return *v*

  onde *L* é o rótulo que identifica a sub-rotina e *v* é o valor de retorno da sub-rotina (se houver tal valor).

Geração de Código Intermediário

Código de três endereços: definição de sub-rotinas

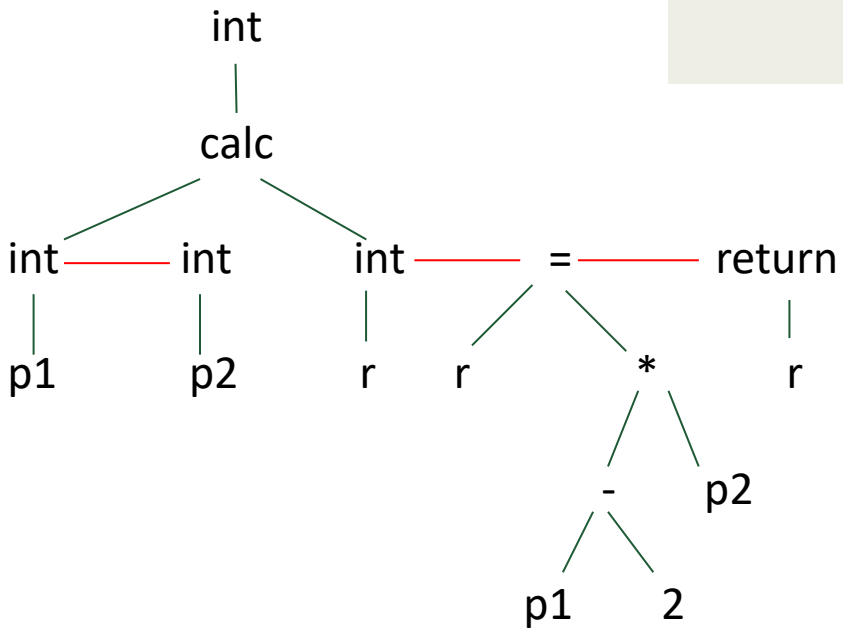
Exemplo (em C-):

```
x = calc(a, b);
.
.
.
int calc(int p1, int p2)
 {
    int r;
    r = (p1 -2) * p2;
    return r;
}
```

[ativação]

[definição]

```
param a
param b
x  =  call  calc, 2
⋮
calc:       _t1 = p1 – 2
            r = _t1 * p2
            return r
```



Geração de Código Intermediário

Código de três endereços: definição de sub-rotinas

Considere a GLC abaixo:

programa → *decl-lista exp*
decl-lista → *decl-lista decl* | ε
decl → **fn id** (*param-lista*) = *exp*
param-lista → *param-lista*, **id** | **id**
exp → *exp* + *exp* | *ativação* | **num** | **id**
ativação → **id** (*arg-lista*)
arg-lista → *arg-lista*, *exp* | *exp*

A entrada abaixo é válida?

```
fn f(x)=2+x
fn g(x,y)=f(x)+y
g(3,4)
```

Geração de Código Intermediário

Código de três endereços: definição de sub-rotinas

Possível definição para o nó da árvore sintática para a GLC:

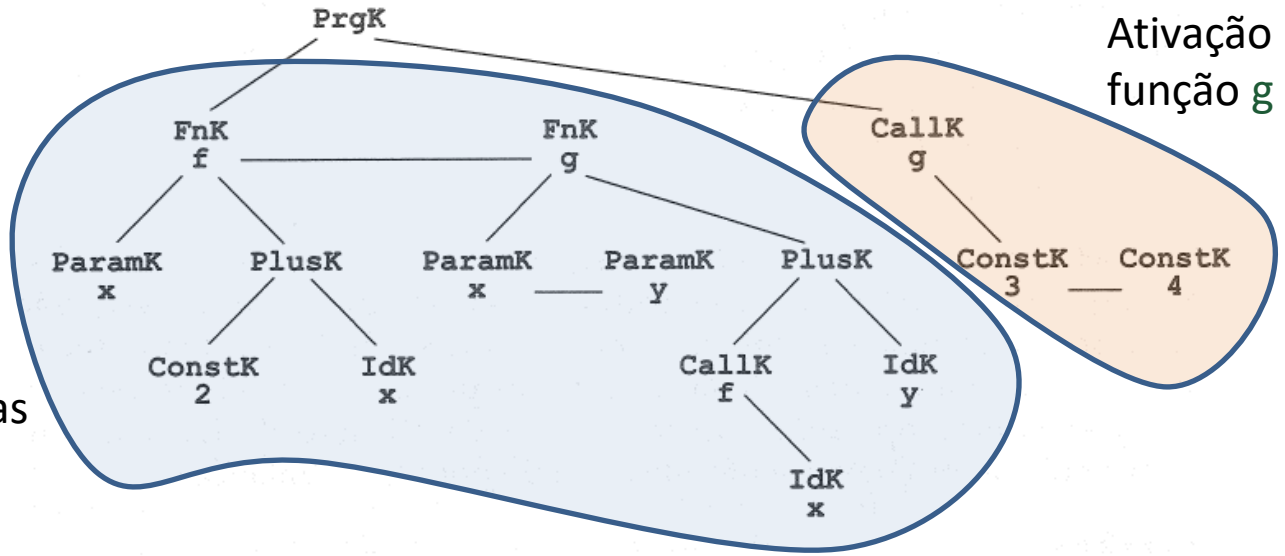
programa → *decl-lista exp*
decl-lista → *decl-lista decl* | ε
decl → **fn id** (*param-lista*) = *exp*
param-lista → *param-lista*, **id** | **id**
exp → *exp* + *exp* | *ativação* | **num** | **id**
ativação → **id** (*arg-lista*)
arg-lista → *arg-lista*, *exp* | *exp*

```
typedef enum
  {PrgK,FnK,ParamK,PlusK,CallK,ConstK,IdK}
  NodeKind;
typedef struct streenode
        { NodeKind kind;
          struct streenode *lchild,*rchild,
                           *sibling;
          char * name; /* usado com FnK,ParamK,
                          CallK,IdK */
          int val; /* usado com ConstK */
        } STreeNode;
typedef STreeNode *SyntaxTree;
```

Geração de Código Intermediário

Código de três endereços: definição de sub-rotinas

Árvore sintática abstrata



Ativação da função g

Definições das funções f e g

Figura 8.13 Árvore sintática abstrata para o exemplo do programa anterior.

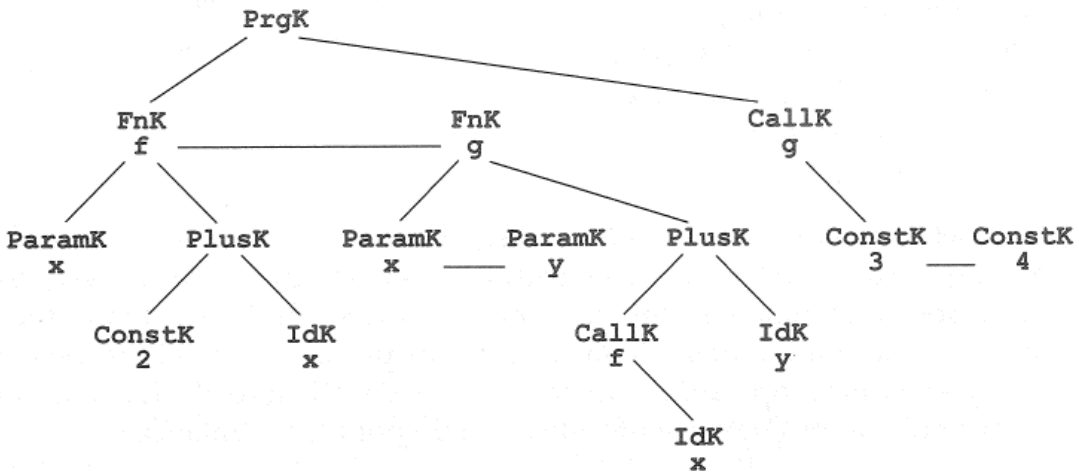
```
fn f(x)=2+x
fn g(x,y)=f(x)+y
g(3,4)
```

Geração de Código Intermediário

Código de três endereços: definição de sub-rotinas

Código de três endereços

```
f:      _t1 = 2 + x
        return _t1
g:      param x
        _t2 = call f, 1
        return _t2 + y
init:   param 3
        param 4
        call g, 2
```




```
fn f(x)=2+x
fn g(x,y)=f(x)+y
g(3,4)
```

Geração de Código Intermediário

Código de três endereços: acesso indexado

```
x = y[i]
```

A posição do item de informação acessado é definida a partir:

- De um endereço base
- E de um deslocamento (o índice)
  - O deslocamento depende do tipo de dado;
  - Normalmente é calculado em bytes.

  Se z é um arranjo de inteiros de 4 bytes, uma referência à z[5] leva à posição que está 20 bytes adiante da posição de z.

Geração de Código Intermediário

Código de três endereços: acesso indexado

Assume o seguinte formato:

```
_t = i * n
v[_t]
```

onde

_t é a variável temporária que armazena o deslocamento calculado em bytes;

i é o índice do vetor;

n é o tamanho em bytes ocupado pelo tipo de dado;

v é a variável indexada.

Geração de Código Intermediário

Código de três endereços: acesso indexado

Exemplo:

Considere que uma variável do tipo inteiro ocupa 4 bytes

y = x[i]

Código de três endereços:

_t1 = i * 4

y = x[_t1]

Se o endereço base de x[] é 100, então o conteúdo acessível por x[_t1] está no endereço 100 + (i * 4)

Exemplo:

Se i = 5, então o endereço de x[_t1] é 120.

Geração de Código Intermediário

Código de três endereços: acesso indexado

Exemplo:

a[i] = b[i] + c[i]

Código de três endereços:

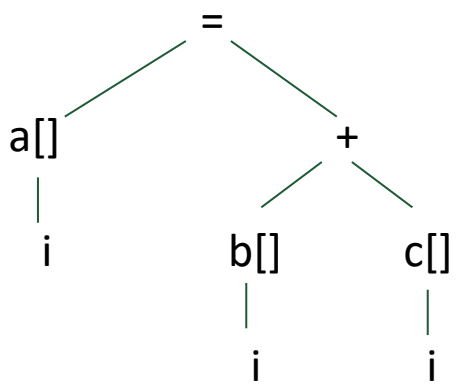
_t1 = i * 4

_t2 = b[_t1]

_t3 = c[_t1]

_t4 = _t2 + _t3

a[_t1] = _t4



Geração de Código Intermediário

```
typedef enum {StmtK,ExpK} NodeKind;
typedef enum {IfK,RepeatK,AssignK,ReadK,WriteK}
             StmtKind;
typedef enum {OpK,ConstK,IdK} ExpKind;

/* ExpType é utilizado para verificação de tipos */
typedef enum {Void,Integer,Boolean} ExpType;

#define MAXCHILDREN 3

typedef struct treeNode
   {  struct treeNode * child[MAXCHILDREN];
      struct treeNode * sibling;
      int lineno;
      NodeKind nodekind;
      union { StmtKind stmt; ExpKind exp;} kind;
      union { TokenType op;
              int val;
              char * name; } attr;
      ExpType type; /* para verificação de tipos de expressões */
   } TreeNode;
```

Figura 3.7 Declarações em C para um nó de árvore sintática TINY.

Estrutura do nó da árvore para linguagem *Tiny*, que pode ser usada como modelo para C-

Geração de Código Intermediário

- Uma representação útil para códigos de três endereços é denominada quádrupla;

- Formato da quádrupla:

  (operação, operando1, operando2, resultado).

Geração de Código Intermediário

```
{ Programa de exemplo
  na linguagem TINY --
  computa o fatorial
}
read x; { inteiro de entrada }
if 0 < x then { não computa se x <= 0 }
  fact := 1;
  repeat
    fact := fact * x;
    x := x - 1
  until x = 0;
  write fact { fatorial de x como saída }
end
```

```
read x
t1 = x > 0
if_false t1 goto L1
fact = 1
label L2
t2 = fact * x
fact = t2
t3 = x - 1
x = t3
t4 = x == 0
if_false t4 goto L2
write fact
label L1
halt
```

Código de três endereços

```
(rd,x,_,_)
(gt,x,0,t1)
(if_f,t1,L1,_)
(asn,1,fact,_)
(lab,L2,_,_)
(mul,fact,x,t2)
(asn,t2,fact,_)
(sub,x,1,t3)
(asn,t3,x,_)
(eq,x,0,t4)
(if_f,t4,L2,_)
(wri,fact,_,_)
(lab,L1,_,_)
(halt,_,_,_)
```

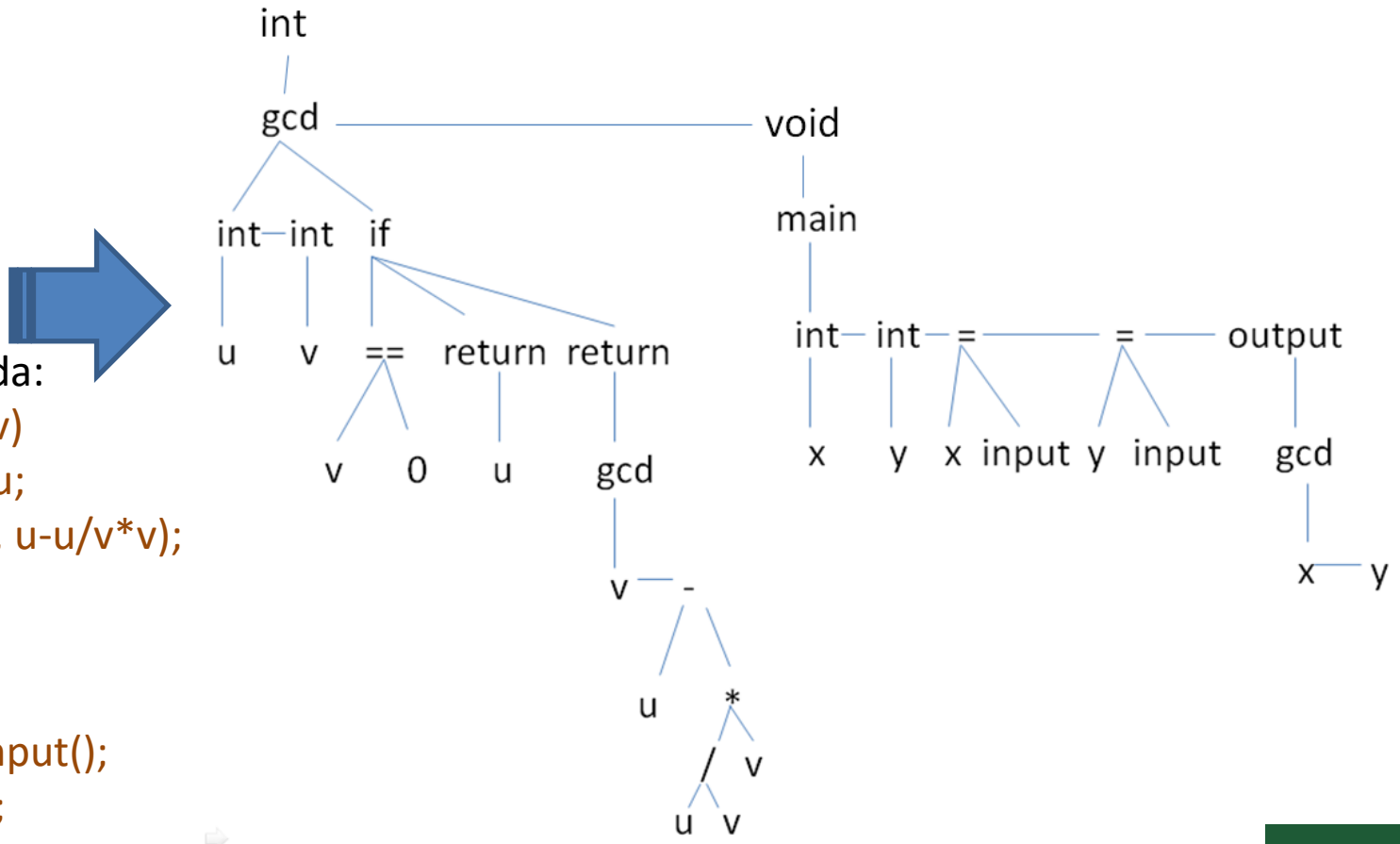
Código de três endereços representado em quádruplas

Geração de Código Intermediário

Árvore sintática do programa para cálculo do gcd (Louden, apêndice A)

Programa de entrada:

```
1  int gcd(int u, int v)
2 { if (v==0) return u;
3   else return gcd(v, u-u/v*v);
4 }
5   void main(void)
6 { int x; int y;
7    x = input(); y = input();
8    output(gcd(x,y));
9 }
```



Geração de Código Intermediário

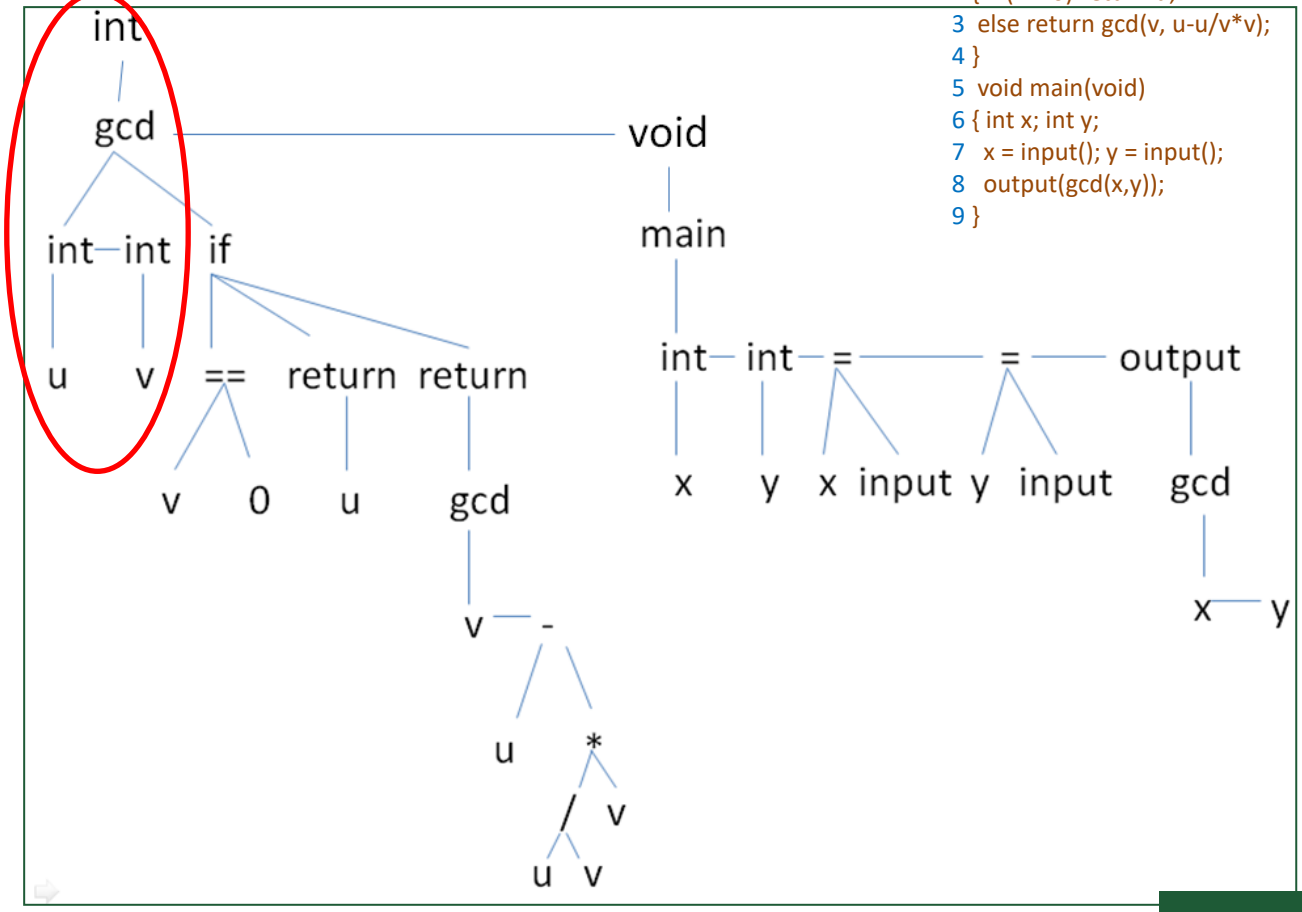
Árvore sintática do programa para cálculo do gcd (Louden, apêndice A)

Programa de entrada:

```
1  int gcd(int u, int v)
2 { if (v==0) return u;
3  else return gcd(v, u-u/v*v);
4 }
5  void main(void)
6 { int x; int y;
7   x = input(); y = input();
8   output(gcd(x,y));
9 }
```

Quádruplas

(FUN, int, gcd, -)
(ARG, int, u, gcd)
(ARG, int, v, gcd)
(LOAD, $t0, u, -)
(LOAD, $t1, v, -)



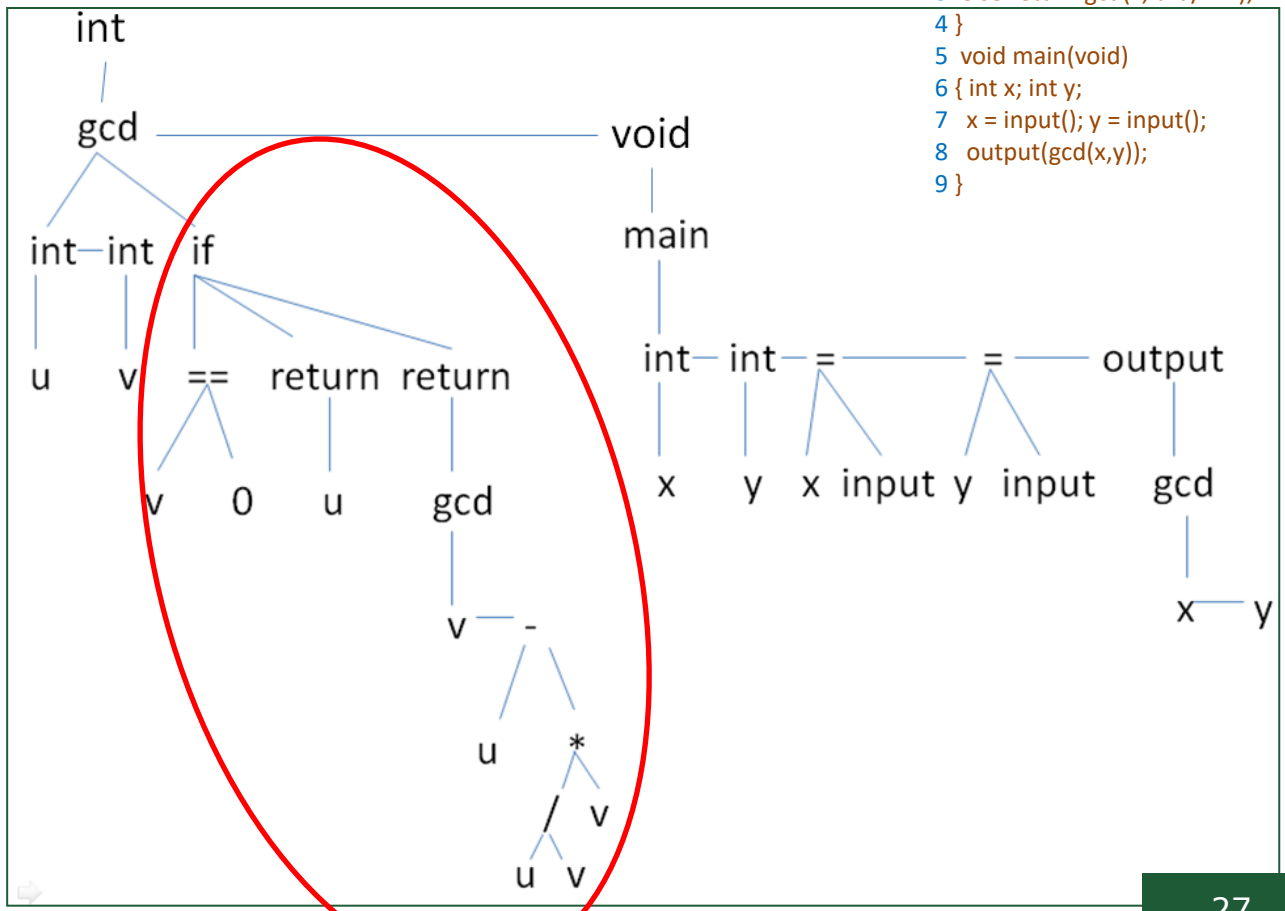
Geração de Código Intermediário

Quádruplas

(EQUAL, $t1, 0, $t2)
(IFF, $t2, L0, -)
(LOAD, $t3, u, -)
(RET, $t3, -, -)
(GOTO, L1, -, -)
(LAB, L0, -, -)
(LOAD, $t4, v, -)
(PARAM, $t4, -, -)
(LOAD, $t5, u, -)
(LOAD, $t6, u, -)
(LOAD, $t7, v, -)
(DIV, $t6, $t7, $t8)
(LOAD, $t9, v, -)
(MUL, $t8, $t9, $t10)
(SUB, $t5, $t10, $t11)
(PARAM, $t11, -, -)
(CALL, gcd, 2, $t12)
(RET, $t12, -, -)
(GOTO, L1, -, -)
(LAB, L1, -, -)
(END, gcd, -, -)

Programa de entrada:

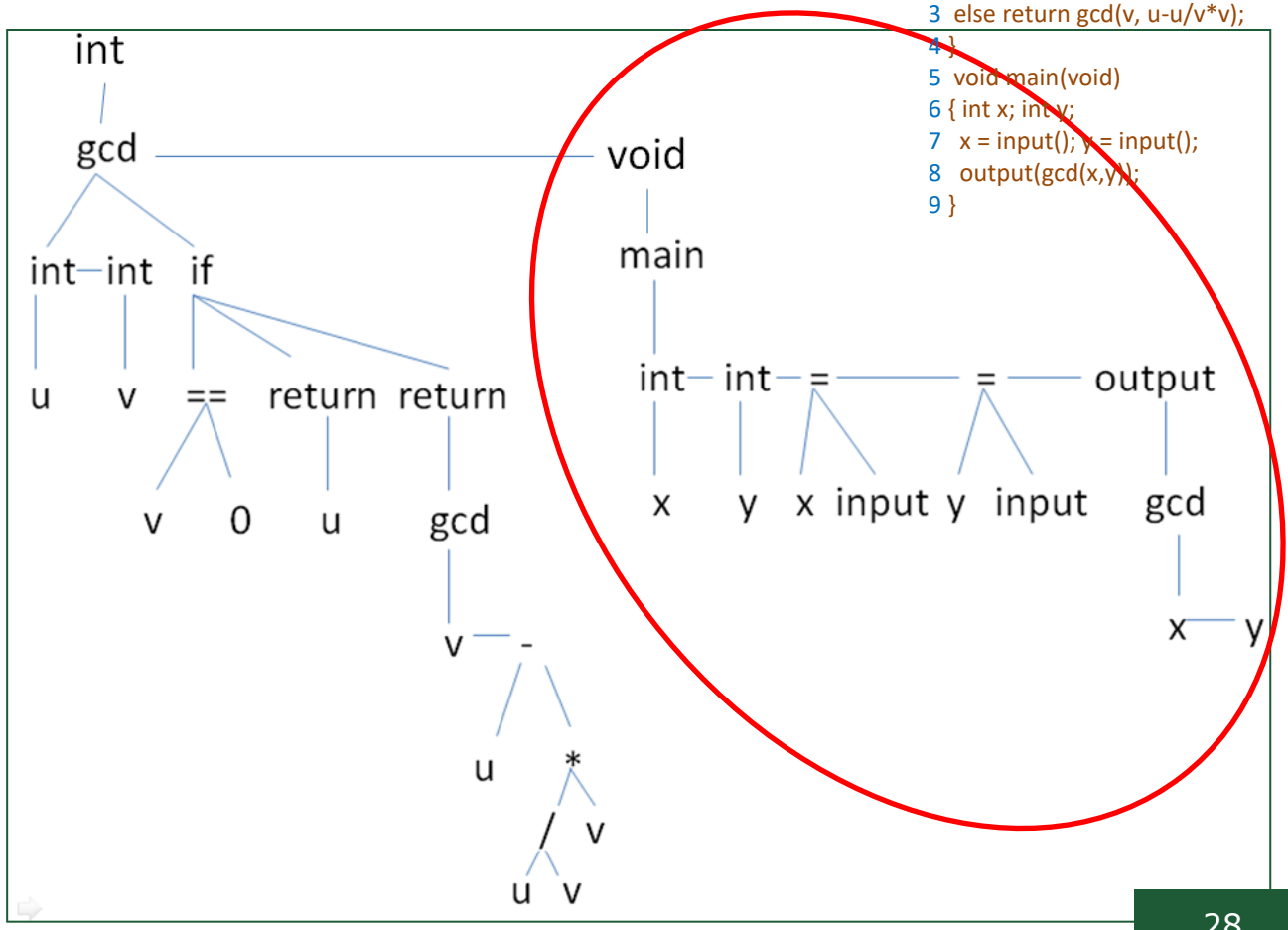
```
1  int gcd(int u, int v)
2 { if (v==0) return u;
3  else return gcd(v, u-u/v*v);
4 }
5  void main(void)
6 { int x; int y;
7   x = input(); y = input();
8   output(gcd(x,y));
9 }
```



Geração de Código Intermediário

Quádruplas

(FUN, void, main, -)
(ALLOC, x, main, -)
(ALLOC, y, main, -)
(LOAD, $t13, x, -)
(CALL, input, 0, $t14)
(ASSIGN, $t13, $t14, -)
(STORE, x, $t13, -)
(LOAD, $t15, y, -)
(CALL, input, 0, $t16)
(ASSIGN, $t15, $t16, -)
(STORE, y, $t15, -)
(LOAD, $t16, x, -)
(PARAM, $t16, -, -)
(LOAD, $t17, y, -)
(PARAM, $t17, -, -)
(CALL, gcd, 2, $t18)
(PARAM, $t18, -, -)
(CALL, output, 1, $t19)
(END, main, -, -)
(HALT, -, -, -)



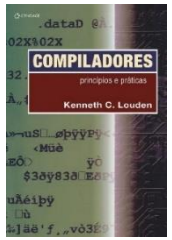
Programa de entrada:

```
1  int gcd(int u, int v)
2 { if (v==0) return u;
3   else return gcd(v, u-u/v*v);
4 }
5  void main(void)
6 { int x; int y;
7    x = input(); y = input();
8    output(gcd(x,y));
9 }
```

Geração de Código Intermediário

Bibliografia consultada

RICARTE, I. **Introdução à Compilação.** Rio de Janeiro: Editora Campus/Elsevier, 2008.



LOUDEN, K. C. **Compiladores: princípios e práticas.** São Paulo: Pioneira Thompson Learning, 2004.

Agradeço ao Professor Luiz Eduardo G. Martins pelo material disponibilizado.